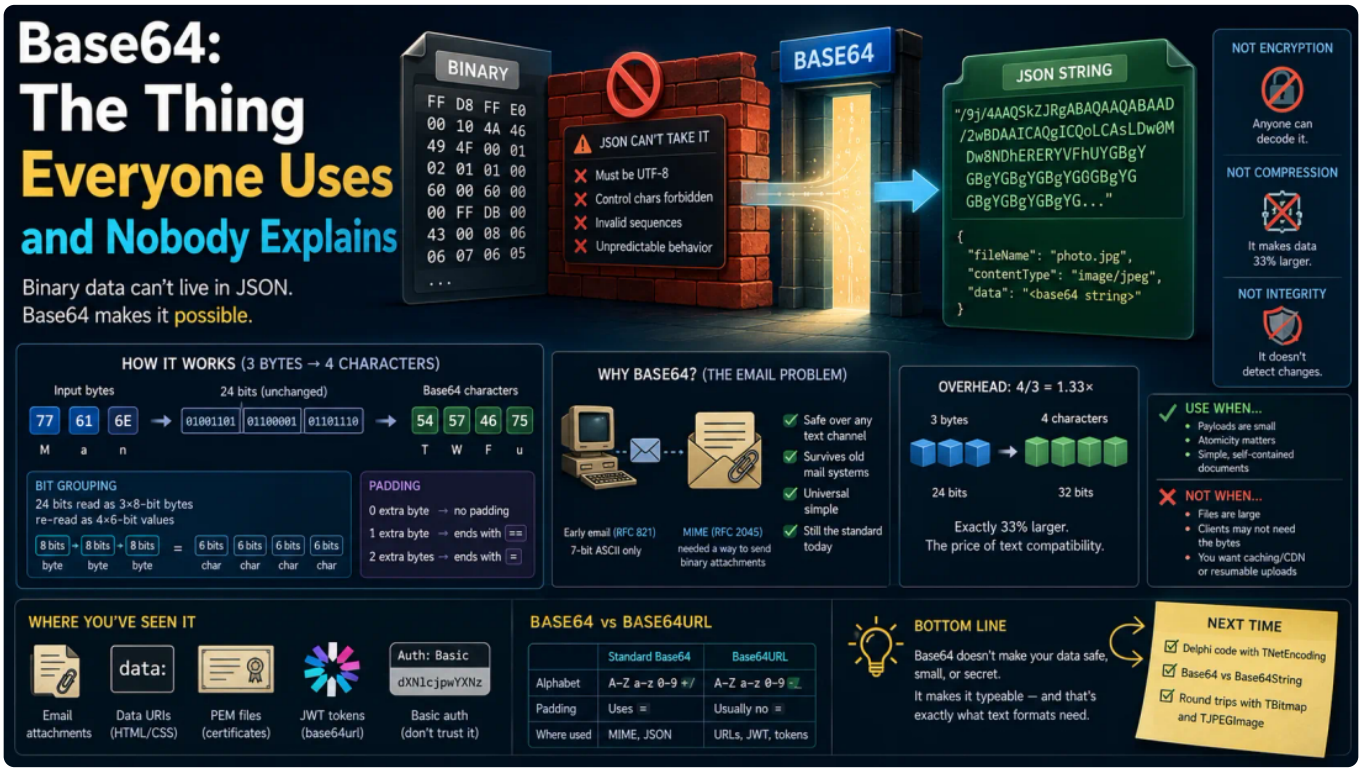
NEWS · JULY 23, 2026

# Base64: The Thing Everyone Uses and Nobody Explains

Part five of the JSON series for Delphi developers — why binary data can't go in JSON, what base64 actually does to your bytes, why email solved this problem first, and what the 33 percent really costs you.

By Dr. Holger Flick




By now this series has established what JSON is made of — six value types and one recursive rule — and we've spent a whole post on the first thing that list doesn't include: dates. The fix there was a convention. A date becomes a string in an agreed format, and both ends have to know the agreement.

Now for the second missing type, and this one is harder. **Binary data.** A JPEG. A PDF invoice. A signed certificate. A ZIP archive. None of these is a string, a number, a boolean, or null, and no amount of convention makes a JPEG into a number.

The answer the whole industry uses is base64, and here's my honest observation from years of Delphi code review: most developers who use it correctly could not tell you what it does. They know the incantation — `TNetEncoding.Base64.EncodeBytesToString` — they know it produces a long string of letters and digits, and they know it works. Ask what actually happens to the bytes and the answers get vague, and I've heard "it compresses it" and "it encrypts it" more than once. Both are wrong, and the second one is dangerous.

So this post is the explanation. What binary data is, why JSON genuinely cannot carry it, what base64 does to your bytes at the bit level, why your email client has been doing this since the early nineties, and what it costs.

No Delphi code at all this time — that's the next post. This one is about understanding the thing well enough that the code is obvious.

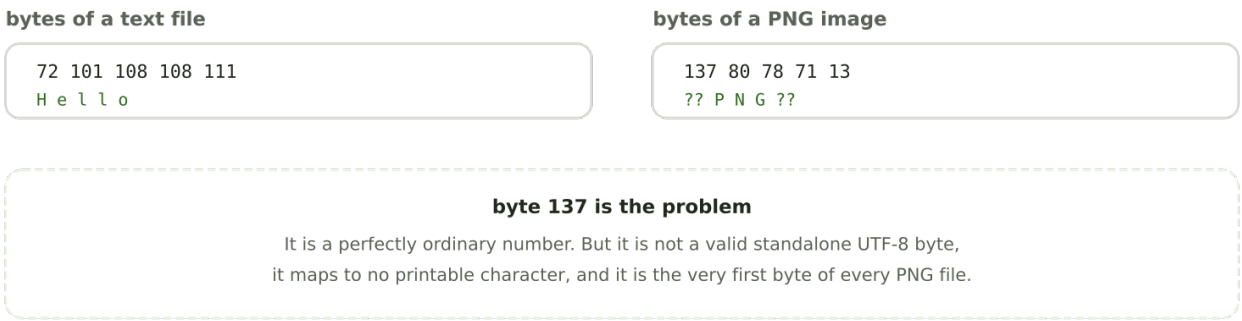
## What "binary" actually means

Let's start lower than feels necessary, because this is where the confusion begins.

Every file on your disk is a sequence of bytes, and a byte is a number from 0 to 255. That's true of a text file and equally true of a JPEG. The difference isn't in the storage — it's in the **agreement about what the numbers mean**.

In a text file, the bytes are meant to be read through a character encoding: a lookup table saying "byte 72 means the letter H." In a JPEG, byte 72 doesn't mean a letter. It might be part of a colour value, a table index, or a fragment of compressed image data. It means whatever the JPEG specification says it means at that position.

So "binary data" isn't a special kind of byte. **It's bytes whose meaning is not text** — and critically, bytes that take *any* value from 0 to 255, including the ones that no character encoding maps to anything printable.




Same byte values, two different agreements about meaning

The dashed box is the crux of the entire post. Byte 137 is not exotic or corrupt — it's just a number. But there is no way to put it directly into a JSON string, and the next section explains exactly why.

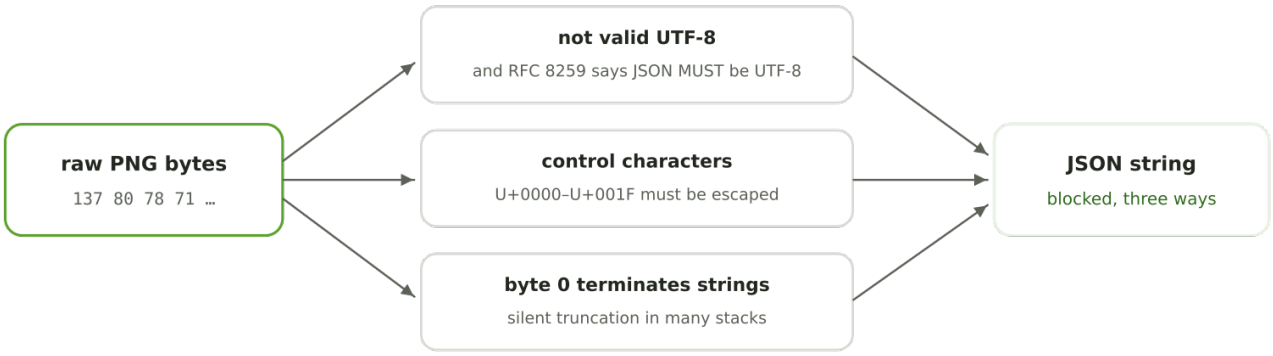
## Why JSON can't take it

Three separate obstacles stand in the way, and each one alone would be enough.

**JSON is text, and text means UTF-8.** RFC 8259 is unambiguous: "JSON text exchanged between systems that are not part of a closed ecosystem MUST be encoded using UTF-8." That's a MUST, the strongest word the IETF has. And UTF-8 is not a free-for-all — it's a structured encoding with strict rules about which byte sequences are legal. Most byte sequences that occur naturally in a JPEG are **not valid UTF-8**. Not "unusual" — actually invalid, rejected by a conforming parser.

**Control characters are forbidden outright.** Even setting UTF-8 aside, JSON strings cannot contain raw control characters — the RFC requires escaping for U+0000 through U+001F. Binary data is full of them. Byte 0 in particular is completely ordinary in a binary file and completely illegal, raw, in a JSON string. (Worse for Delphi and C developers: a zero byte terminates a string in many languages, so even code that *tolerated* it would truncate silently.)

**Invalid sequences produce unpredictable behaviour.** RFC 8259 is candid that when a JSON text contains bit sequences that can't encode Unicode characters, "the behavior of software that receives JSON texts containing such values is unpredictable." Different parsers, different results, no guarantee. That's not a bug to work around; it's the spec declining to define the case.




Three independent reasons the raw bytes cannot go in

Notice that these are three *independent* barriers. Solve any one and the other two remain. That's why the solution isn't a workaround or an escape trick — it has to be a genuine change of representation.

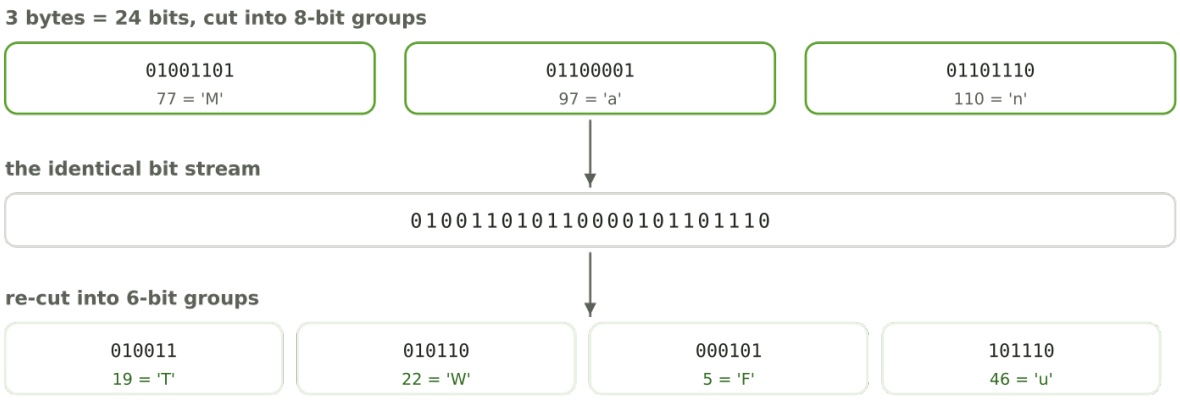
## What base64 actually does

Here is the idea in one sentence, and it's worth reading twice: **base64 rewrites your bytes using only 64 characters that are safe everywhere, by regrouping the bits.**

Not compressing. Not encrypting. **Regrouping bits.** That's the whole mechanism, and once you see it the 33% overhead stops being a mystery and becomes arithmetic.

A byte is 8 bits. There are 256 possible values, and most of them are not safe as text. So base64 asks a different question: what if we chopped the bit stream into **6-bit** pieces instead of 8-bit ones? Six bits gives 2v = 64 possible values — and 64 distinct, printable, universally-safe characters are easy to find. RFC 4648 defines the alphabet: `A`–`Z`, `a`–`z`, `0`–`9`, `+`, and `/`.

The regrouping works on 3 bytes at a time, because 3 bytes is 24 bits, and 24 divides evenly by both 8 and 6:




Three bytes in, four characters out — the same 24 bits, cut differently

Follow the middle row and you'll see the point: **the bits never change.** `Man` becomes `TWFu`, and the bit string in the middle is identical in both readings. Three bytes went in; four characters came out. All that changed is where the scissors fell.

That's also where the overhead comes from, with no hand-waving needed: 4 output characters for every 3 input bytes is exactly 4/3, a **33% increase**. RFC 2045 states it plainly — encoded data are "consistently only about 33 percent larger than the unencoded data."

## The equals signs at the end

Now the detail everyone has seen and few can explain: why do base64 strings so often end in `=` or `==`?

The scheme works in 3-byte groups, but real data doesn't come in convenient multiples of three. If your input ends with 1 or 2 leftover bytes, the encoder pads the bits out to a full group and then appends `=` characters to record how much of that final group was real:

- Input length divides by 3 ’ no padding.
- **1 byte** left over ’ output ends`==`.
- **2 bytes** left over ’ output ends`=`.

The `=` is the 65th character in the RFC's alphabet, and it carries **no data at all** — it's a length marker so the decoder knows how many bytes to produce from the final group. RFC 4648 requires it by default: "Implementations MUST include appropriate pad characters at the end of encoded data unless the specification referring to this document explicitly states otherwise." That escape clause matters and reappears below.

> **Base64 is not security. Not even slightly.**
> This is the misconception with actual consequences, so let me be blunt about it. Base64 is **not encryption**.
>
> There is no key, no secret, and nothing to break — decoding is a public, mechanical operation that any programmer, browser console, or command-line tool performs instantly. A base64 string is *exactly as secret* as the bytes inside it. It is also **not compression**. It makes data 33% larger, not smaller. And it is **not integrity protection** — it's not a checksum or a signature, and it will not tell you if bytes were altered in transit. Where this actually hurts: HTTP Basic authentication base64-encodes `username:password`, which looks encoded and is therefore mistaken for protected. It is not. It's plaintext with extra steps, which is

> precisely why Basic auth over anything but HTTPS is a credential leak. If you need secrecy, encrypt; if you need integrity, sign. Base64 solves a *transport* problem, and only that one.

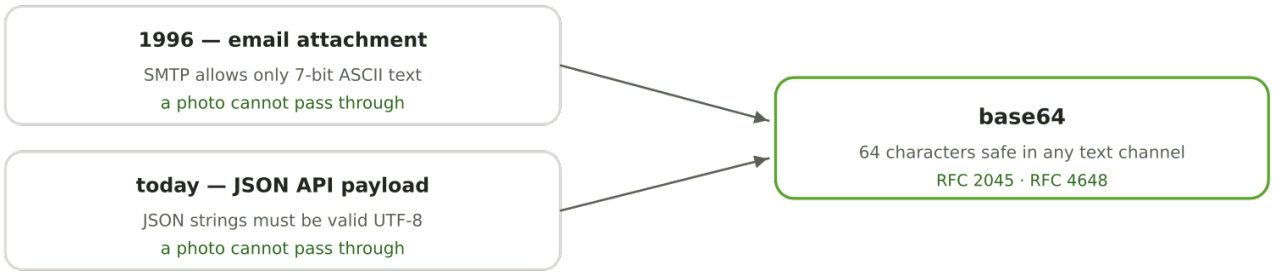
## Email solved this first

The reason base64 feels so universal is that it is — and it got that way solving this exact problem more than a decade before JSON existed.

Early internet email was specified for text only. RFC 821, the original SMTP standard from 1982, restricted messages to 7-bit US-ASCII. Not 8-bit bytes — **seven bits per character**, because that's what the mail infrastructure of the era was built on. Mail servers along the route could and did mangle the eighth bit, strip characters, and rewrite line endings.

Sending a photograph through that was impossible. So MIME — Multipurpose Internet Mail Extensions, 1996 — defined a set of content-transfer-encodings, and base64 was the one for arbitrary binary data. RFC 2045 describes the design goal exactly: an encoding "designed to represent arbitrary sequences of octets in a form that need not be humanly readable," transforming data "into material in the '7bit' range, thus making it safe to carry over restricted transports."

That phrase — *need not be humanly readable* — is the honest one. Base64 was never meant to be pretty. It was meant to survive.




The same problem, twenty years apart, with the same answer

The diagram is the argument for learning this properly rather than memorising an incantation. It is **the same problem**: a channel that only accepts text, and data that isn't text. Understanding it once means you also understand why data URIs work in HTML, why JWT tokens look the way they do, why PEM certificate files are readable ASCII, and why `Authorization: Basic` headers have that shape. All the same trick.

> **The variant you'll meet: base64url**
> One wrinkle worth knowing before it bites you. Standard base64 uses `+` and `/`, and both have special meaning in URLs — `/` is a path separator, and `+` is often read as an encoded space. So RFC 4648 defines a **URL-safe variant**, base64url, that swaps them for `-` and `_`. It also commonly drops the `=` padding, using the escape clause quoted earlier. This is not cosmetic: a base64url string fed to a standard base64 decoder will fail or produce garbage, and vice versa. If you work with JWT tokens you are already using base64url — it's what the spec mandates — which is why JWTs contain `-` and `_` and usually have no trailing `=`. Delphi exposes both, and the next post shows exactly where.

## What the 33 percent really costs

Time for the honest engineering discussion, because "base64 adds 33%" gets repeated as though it settles something, and it doesn't.

The overhead is real and it compounds. A 5 MB photo becomes about 6.7 MB of base64. But the encoded size is not the whole cost. That string has to be **built in memory** by the sender, **held in memory** while the JSON document is assembled around it, transmitted, then **parsed and decoded** by the receiver — and a naive implementation may hold several copies at once. For a large file, peak memory can be a multiple of the original, which is a far more common production problem than the bandwidth.

There's also the CPU cost of encoding and decoding, though for most workloads that's genuinely negligible next to the network time.

So when is embedding binary in JSON the right call?

**It's a good fit when** the payload is small (thumbnails, icons, signatures, small certificates), when atomicity matters — you want the metadata and the bytes to arrive as one indivisible unit that can't half-fail — or when a single self-contained document is simply worth more than the efficiency, as in a webhook or a message queue payload.

**It's a poor fit when** the files are large, when clients may want the metadata without the bytes, when you'd like to cache or CDN the binary separately, or when you need resumable uploads.

> **The alternative that usually wins for large files**
> For anything substantial, the better pattern is almost always to **not put the bytes in the JSON at all**. Upload the file separately — a plain `POST` of the raw bytes, a `multipart/form-data` request, or a pre-signed URL straight to object storage — and put a **URL in the JSON** instead. No 33% overhead, no memory spike, the binary is cacheable, and clients fetch it only if they need it. My rule of thumb, offered as a rule of thumb and not a law: under a few hundred kilobytes, embedding is usually fine and the simplicity is worth it; into the megabytes, reach for a URL. But the threshold genuinely depends on your traffic and constraints — a low-volume internal service embedding 5 MB payloads may be perfectly healthy, while a high-traffic endpoint doing it at 200 KB might not be. Measure your own case rather than trusting my number.

## The formats that don't have this problem

It would be dishonest to present base64 as the only possible answer, because it isn't — several formats carry binary natively, without any encoding step.

BSON, the binary format MongoDB uses, has a first-class binary type. MessagePack is essentially "JSON but binary" and carries raw bytes directly. CBOR is an IETF standard with the same property. Protocol Buffers has a `bytes` type and is widely used inside large systems, particularly with gRPC. All of them avoid the 33% entirely, and all are real, well-engineered, actively-used technologies.

But here's the practical reality, and it's why this post treats base64 as the answer: **you will rarely meet any of them in a public web API.** Their natural home is inside a system — service-to-service traffic, storage engines, mobile sync protocols — where both ends are controlled by the same team and can agree on a binary format. The moment an API is public, it needs to be consumable from a browser, from `curl`, from a language whose ecosystem may have no library for your chosen format, by a developer who needs to eyeball the response while

debugging. JSON's readability and universality win that argument decisively, and they win it *despite* the 33%, which tells you how much those properties are worth.

So the trade is real and the alternatives are good. It's just that for the specific job of "a public API that anyone can call," JSON with base64 has become the de facto standard — not because it's elegant, but because it works everywhere and needs nothing installed. That's usually how standards happen.

## Takeaways

Base64 is a small idea wearing a mysterious name, and knowing precisely what it does removes a whole category of confusion.

- **Binary data is just bytes whose meaning isn't text** — including the many byte values that no character encoding maps to anything printable.
- **JSON genuinely cannot carry raw bytes.** RFC 8259 says JSON text MUST be UTF-8, control characters must be escaped, and invalid sequences make parser behaviour "unpredictable." Three independent barriers, one required solution.
- **Base64 regroups bits, nothing more.** Twenty-four bits read as three 8-bit bytes, re-read as four 6-bit values, each mapped to one of 64 safe characters. That 4-for-3 ratio *is* the 33% overhead. The trailing `=` signs are padding, not data.
- **It is not encryption, compression, or integrity protection.** A base64 string is exactly as secret as its contents — which is why Basic auth over plain HTTP leaks credentials.
- **Email solved this in 1996 and the reasoning transfers unchanged.** Same problem, same answer — and it's why data URIs, PEM files, and JWTs all look the way they do.
- **Embed small payloads; link to large ones.** Under a few hundred KB, base64 in JSON is usually fine. Into the megabytes, put a URL in the JSON and move the bytes separately.

> Base64 doesn't make your data safe, small, or secret. It makes it *typeable* — and that turns out to be exactly the property a text format needs.

Next time we write the code. Encoding and decoding with `TNetEncoding`, the difference between `Base64` and `Base64String` that silently breaks JSON payloads, and full round trips with real images — `TBitmap` and `TJPEGImage` — using nothing but the RTL.

INDEPENDENT · AD-FREE · READER-SUPPORTED

# Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

## Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

## Ways to support

KO-FI

### Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

### Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

### Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

**Support on Ko-fi —**